Issues and Comments about Object Oriented Technology in Aviation

Issue #	Topic	Issue Statement
1	Dead/ deactivated code	Deactivated Code will be found in any application that uses general purposed libraries or object-oriented frameworks. (Note that this is the case where unused code is NOT removed by smart linkers.)
2	Dynamic binding/ dispatch	Flow Analysis, recommended for Levels A-C, is complicated by Dynamic Dispatch (just which method in the inheritance hierarchy is going to be called?).
3	Dynamic binding/ dispatch	Timing Analysis, recommended for Levels A-D is complicated by Dynamic Dispatch (just how much time will be expended determining which method to call?).
4	Dynamic binding/ dispatch	Requirements Testing, recommended for Levels A-D, and Structural Coverage Analysis, recommended for Levels A-C, are complicated by Inheritance, Overriding and Dynamic Dispatch (just how much of the existing verification of the parent class can be reused in its subclasses?).
5	Dynamic binding/ dispatch	Structural Coverage Analysis, recommended for Levels A-C, is complicated by Dynamic Dispatch (just which method in the inheritance hierarchy does the execution apply to?).
6	Dynamic binding/ dispatch	Conformance to the guidelines in DO-178B concerning traceability from source code to object code for Level A software is complicated by Dynamic Dispatch (how is a dynamically dispatched call represented in the object code?).
7	Dynamic binding/ dispatch	Polymorphic, dynamically bound messages can result in code that is error prone and hard to understand.
8	Dynamic binding/ dispatch	Dynamic dispatch presents a problem with regard to the traceability of source code to object code that requires “additional verification” for level A systems as dictated by DO-178B section 6.4.4.2b.
9	Dynamic binding/ dispatch	Dynamic dispatch complicates flow analysis, symbolic analysis, and structural coverage analysis.
10	Dynamic binding/ dispatch	Inheritance, polymorphism, and linkage can lead to ambiguity.

11	Dynamic binding/ dispatch	The use of inheritance and polymorphism may cause difficulties in obtaining structural coverage, particularly decision coverage and MC/DC
12	Dynamic binding/ dispatch	Source to object code correspondence will vary between compilers for inheritance and polymorphism.
13	Dynamic binding/ dispatch	Polymorphic and overloaded functions may make tracing and verifying the code difficult.
14	Inheritance	Requirements Testing, recommended for Levels A-D, and Structural Coverage Analysis, recommended for Levels A-C, are complicated by Inheritance, Overriding and Dynamic Dispatch (just how much of the existing verification of the parent class can be reused in its subclasses?).
15	Inheritance	Multiple interface inheritance can introduce cases in which the developer's intent is ambiguous. (when the same definition is inherited from more than one source is it intended to represent the same operation or a different one?)
16	Inheritance	Flow Analysis and Structural Coverage Analysis, recommended for Levels A-C, are complicated by Multiple Implementation Inheritance (just which of the inherited implementations of a method is going to be called and which of the inherited implementations of an attribute is going to be referenced?). The situation is complicated by the fact that inherited elements may reference one another and interact in subtle ways which directly affect the behavior of the resulting system.
17	Inheritance	Use of inheritance (either single or multiple) raises issues of compatibility between classes and subclasses.
18	Inheritance	Inheritance and overriding raise a number of issues with respect to testing: "Should you retest inherited methods? Can you reuse superclass tests for inherited and overridden methods? To what extent should you exercise interaction among methods of all superclasses and of the subclass under test?"
19	Inheritance	Inheritance can introduce problems related to initialization. "Deep class hierarchies [in particular] can lead to initialization bugs." There is also a risk that a subclass method will be called (via dynamic dispatch) by a higher level constructor before the attributes associated with the subclass have been initialized.

20	Inheritance	"A subclass-specific implementation of a superclass method is [accidentally] omitted. As a result, that superclass method might be incorrectly bound to a subclass object, and a state could result that was valid for the superclass but invalid for the subclass owing to a stronger subclass invariant. For example, Object-level methods like isEqual or copy are not overridden with a necessary subclass implementation".
21	Inheritance	"A subclass [may be] incorrectly located in a hierarchy. For example, a developer locates SquareWindow as a subclass of RectangularWindow, reasoning that a square is a special case of a rectangle ... Suppose that [the method] resize(x, y) is inherited by SquareWindow. It allows different lengths for adjacent sides, which causes SquareWindow to fail after it has been resized. This situation is a design problem: a square is not a kind of a rectangle, or vice versa. Instead both are kinds of four-sided polygons. The corresponding design solution is a superclass FourSidedWindow, of which RectangularWindow and SquareWindow are subclasses."
22	Inheritance	"A subclass either does not accept all messages that the superclass accepts or leaves the object in a state that is illegal in the superclass. This situation can occur in a hierarchy that should implement a subtype relationship that conforms to the Liskov substitution principle."
23	Inheritance	"A subclass computes values that are not consistent with the superclass invariant or superclass state invariants."
24	Inheritance	"Top-heavy multiple inheritance and very deep hierarchies (six or more subclasses) are error-prone, even when they conform to good design practice. The wrong variable type, variable, or method may be inherited, for example, due to confusion about a multiple inheritance structure"
25	Inheritance	The ability of a subclass to directly reference inherited attributes tightly couples the definitions of the two classes.
26	Inheritance	Inheritance can be abused by using it as a "kind of code-sharing macro to support hacks without regard to the resulting semantics"
27	Inheritance	When the same operation is inherited by an interface via more than one path through the interface hierarchy (repeated

		inheritance), it may be unclear whether this should result in a single operation in the subinterface, or in multiple operations.
28	Inheritance	When a subinterface inherits different definitions of the same operation [as a result of redefinition along separate paths], it may be unclear whether/how they should be combined in the resulting subinterface.
29	Inheritance	Use of multiple inheritance can lead to “name clashes” when more than one parent *independently* defines an operation with the same signature.
30	Inheritance	When *different* parent interfaces define operations with different names but compatible specifications, it is unclear whether it should be possible to merge them in a subinterface.
31	Inheritance	It is unclear whether the normal overload resolution rules should apply between operations inherited from different superinterfaces or whether they should not (as in C++).
32	Inheritance	It is important that the overriding of one operation by another and the joining of operations inherited from different sources always be intentional rather than accidental.
33	Inheritance	Multiple inheritance complicates the class hierarchy
34	Inheritance	Multiple inheritance complicates configuration control
35	Inheritance	When inheritance is used in the design, special care must be taken to maintain traceability. This is particularly a concern if multiple inheritance is used.
36	Inheritance	Source to object code correspondence will vary between compilers for inheritance and polymorphism.
37	Inheritance	Overuse of inheritance, particularly multiple inheritance, can lead to unintended connections among classes, which could lead to difficulty in meeting the DO-178B/ED-12B objective of data and control coupling.
38	Inheritance	Multiple inheritance should be avoided in safety critical, certified systems.
39	Inheritance	“Top-heavy multiple inheritance and very deep hierarchies (six or more subclasses) are error-prone, even when they conform to good design practice. The wrong variable type, variable, or method may be inherited, for example, due to confusion about a multiple inheritance structure”

40	Inheritance	Reliance on programmer specified optimizations of the inheritance hierarchy (invasive inheritance) is potentially error prone and unsuitable for safety critical applications.
41	Inheritance	Inheritance, polymorphism, and linkage can lead to ambiguity.
42	Inheritance	Inheritance allows different objects to be treated in the same general way. Inheritance as used in Object Oriented Technology is combining several like things into a fundamental building block. The programmer is allowed to take a group of these like things and refer to them in a general way. One routine can be used for all types that inherit from the fundamental building block. The more often a programmer can use the generic behavior of the parent, the more productive the programmer is. The problem I see is that the generic behavior will not always be precise enough for all the applications, and that critical judgement is required to determine when the programmer needs to specialize the behavior of one of the object rather than use the generic. Who will issue that critical judgement? Who will find all the instances where the general case is too far away from the precision required?
43	Inlining	Flow Analysis, recommended for levels A-C, is impacted by Inlining (just what are the data coupling and control coupling relationships in the executable code?). The data coupling and control coupling relationships can transfer from the inlined component to the inlining component.
44	Inlining	Stack Usage and Timing Analysis, recommended for levels A-D, are impacted by Inlining (just what are the stack usage and worst-case timing relationships in the executable code?). Since inline expansion can eliminate parameter passing, this can effect the amount of information pushed on the stack as well as the total amount of code generated. This, in turn, can effect the stack usage and the timing analysis.
45	Inlining	Structural Coverage Analysis, recommended for levels A-C, is complicated by Inlining (just what is the “logical” coverage of the inline expansions on the original source code?). This is generally only a problem when inlined code is optimized. If statements are removed from the inlined version of a component, then coverage

		of the inlined component is no longer sufficient to assert coverage of the original source code.
46	Inlining	Conformance to the guidelines in DO-178B concerning traceability from source code to object code for Level A software is complicated by Inlining (is the object code traceable to the source code at all points of inlining/expansion?). Inline expansion may not be handled identically at different points of expansion. This can be especially true when inlined code is optimized.
47	Inlining	Inlining may affect tool usage and make structural coverage more difficult for levels A, B, and C.
48	Structural coverage	The unrestricted use of certain object-oriented features may impact our ability to meet the structural coverage criteria of DO-178B.
49	Structural coverage	Statement coverage when polymorphism, encapsulation or inheritance is used.
50	Templates	Templates are instantiated by substituting a specific type argument for each formal type parameter defined in the template class or operation. Passing a test suit for some but not all instantiations cannot guarantee that an untested instantiation is bug free.
51	Templates	Nested templates, child packages (Ada), and friend classes (C++) can result in complex code and hard to read error messages on many compilers.
52	Templates	Templates can be compiled using "code sharing" or "macro-expansion". Code sharing is highly parametric, with small changes in actual parameters resulting in dramatic differences in performance. Code coverage, therefore, is difficult and mappings from a generic unit to object code can be complex when the compiler uses the "code sharing" approach.
53	Templates	Macro-expansion can result in memory and timing issues, similar to those identified for inlining.
54	Templates	The use of templates can result in code bloat. Many C++ compilers cause object code to be repeated for each instance of a template of the same type.
55	Tools	How can we meet the structural coverage requirements of DO-178B with respect to dynamic dispatch? There is cause for

		concern because many current Structural Coverage Analysis tools do not “understand” dynamic dispatch, i.e. do not treat it as equivalent to a call to a dispatch routine containing a case statement that selects between alternative methods based on the run-time type of the object.
56	Tools	How can we meet the control and data flow analysis requirements of DO-178B with respect to dynamic dispatch?
57	Tools	How can deactivated code be removed from an application when general purpose libraries and object-oriented frameworks are used but not all of the methods and attributes of the classes are needed by a particular application?
58	Tools	How can we enforce the rules that restrict the use of specific OO features?
59	Other	Implicit type conversion raises certification issues related to source to object code traceability, the potential loss of data or precision, and the ability to perform various forms of analysis called for by [DO-178B] including structural coverage analysis and data and control flow analysis. It may also introduce significant hidden overheads that affect the performance and timing of the application.
60	Other	Overloading can be confusing and contribute to human error when it introduces methods that have the same name but different semantics. Overloading can also complicate matters for tools (e.g., structural coverage and control flow analysis tools) if the overloading rules for the language are overly complex.
61	Other	Loss of traceability due to the translation of functional requirements to an object-oriented design.
62	Other	Functional coverage of the low level requirement
63	Other	Philosophy of Functional Software Engineering - Most of the training, tools and principles associated with software engineering and assurance, including those of RTCA DO-178B, have been focused on a software function perspective, in that there is an emphasis on software requirements and design and verification of those requirements and the resulting design using reviews, analyses, and requirements-based (functional) testing, and RBT coverage and structural coverage analysis.

		Philosophy of Objects and Operations - Although generally loosely and inconsistently defined, OOT focuses on "objects" and the "operations" performed by and/or to those objects, and may have a philosophy and perspective that are not very conducive to providing equivalent levels of design assurance as the current "functional" approach.
64	Other	Software/software integration testing is often avoided. The position defended by the industry is that the high level of interaction between a great number of objects could lead to a combinative explosion of test cases.